Universidade de São Paulo
Instituto de Matemática e Estatística
Bachalerado em Ciência da Computação

Renan Teruo Carneiro
Vitor Cerqueira Santos

Aplicando o Arcabouço OpenTuner a Jogos Digitais

São Paulo
2ª Versão, Janeiro de 2016

Aplicando o Arcabouço OpenTuner a Jogos Digitais

Monografia final da disciplina
MAC0499 – Trabalho de Formatura Supervisionado.

Supervisor: Prof. Dr. Alfredo Goldman
Cosupervisor: Pedro Bruel, Doutorando IME/USP

São Paulo
2ª Versão, Janeiro de 2016

Sumário

Capítulo 1

Introdução

Neste trabalho serão exploradas possíveis aplicações do conceito de *Autotuning*, ou Ajuste Fino, a jogos digitais. Desde otimização de código, até ajustes de comportamento de inteligências artificiais, podem ser feitas com a aplicação deste conceito. Mas, primeiramente, é necessário explicar a ideia de *autotuning*. O *autotuning* parte da proposta de automatizar o processo de otimização de um programa, alterando seus parâmetros ou configurações de algoritmos. Um sistema que realiza essa ação é comumente chamado de *autotuner*, e pode funcionar de várias maneiras. O que ele faz é testar diversas configurações diferentes de parâmetros ou algoritmos, e escolher quais destas obtêm o melhor desempenho para o programa, seja este um melhor resultado ou um menor tempo de execução. Um exemplo seria a escolha de *flags* de compilação que geram um programa com o menor tempo de execução sem introduzir erros (Ansel *et al.* (2014), Bruel *et al.*).

Os possíveis usos e aplicações de um *autotuner* no campo de jogos digitais são diversos, desde a criação de ferramentas externas para jogadores que querem conseguir o máximo de seus pontos em um jogo, até possíveis implementações de meta-ferramentas de análise do jogo a nível de desenvolvimento. Um desenvolvedor poderia utilizar um *autotuner* para otimizar uma seção de seu código, por exemplo. Fazendo isso, pode-se conseguir realizar uma otimização de maneira muito mais rápida, e possivelmente mais efetiva, do que se conseguiria realizando a mesma tarefa de forma manual (Hoos (2012)).

Para este trabalho foi utilizado o OpenTuner, um arcabouço de *autotuning* focado em técnicas que realizam otimização de parâmetros, analisando os resultados gerados pelo programa a ser ajustado e explorando o espaço de busca. O OpenTuner usa diversos algoritmos para navegar pelo espaço de buscas, e eventualmente encontrar a combinação de parâmetros que gera o resultado desejado, ou se aproxima da melhor maneira deste resultado.

O objetivo deste trabalho é avaliar a aplicabilidade de técnicas de ajuste fino em jogos. Para este fim, foram realizados experimentos usando o jogo OpenTTD e um *autotuner* implementado com o OpenTuner que otimiza parâmetros de inteligências artificiais executadas em um mapa de um servidor de OpenTTD. Este jogo consiste de um simulador de uma empresa de gerência de transportes, e é o papel do jogador construir e expandir uma rede de serviços de transporte da maneira mais lucrativa possível. Os experimentos que foram realizados para as IAs (Inteligências Artificiais) foram analisar quais parâmetros afetam o ganho de caixa em um determinado período de tempo. Foram observados fatores como facilidade de implementação, funcionamento adequado e resultados obtidos.

O restante do texto está organizado da seguinte maneira: primeiramente, são apresentadas as fundamentações teóricas sobre Otimização de Parâmetros, o que abre caminho para a apresentação do OpenTuner e de seu funcionamento. São mostrados também os estudos preliminares que levaram à escolha do jogo e do projeto realizado. A seção seguinte mostra o

jogo OpenTTD, explica seus objetivos e detalhes de funcionamento, e como são apresentadas as IAs do jogo. Enfim há a seção em que é detalhada a implementação realizada como teste de aplicabilidade do OpenTuner no jogo escolhido, o OpenTTD. Os experimentos realizados, e seus resultados são apresentados e discutidos, e enfim é dada uma conclusão com base nos dados encontrados.

Capítulo 2

Fundamentação Teórica

2.1 Otimização de Parâmetros

Otimização de parâmetros consiste em encontrar os parâmetros que otimizam uma aplicação. Seja esse objetivo a minimização do tempo de execução, ou algo como a maximização do resultado da computação feita. Existem diversas maneiras de se fazer esta otimização. Uma forma de se realizar isto era por testes e esforço direto do programador, envolvendo dificuldades de implementação, testes planejados, e complexidade teórica. Um desenvolvedor passaria muito tempo lutando contra essas dificuldades, que adicionavam o esforço de se encontrar uma forma otimizada de resolver o problema, sobre a dificuldade de resolver o problema base. A otimização automatizada surgiu de um desejo de se relegar o trabalho de encontrar a melhor maneira de se executar uma tarefa para o computador. Diversos arcabouços e aplicações surgiram, com o propósito de fazer essa otimização de forma automatizada (Hoos (2012)).

Em desenvolvimento de software, sempre existem diferentes maneiras de se resolver um problema encontrado pelo desenvolvedor, seja pelo uso de diferentes algoritmos, diferentes modelagens do problema, ou aproximações alternativas. Quando se está trabalhando em algum problema, normalmente estes diversos métodos são considerados, e os que melhor resolvem o problema são escolhidos e implementados. Essa escolha se baseia em diversos fatores, dentre os quais a eficiência, a corretude e o esforço de implementação são os mais importantes. Devido a esse fato, a solução escolhida é uma que apresenta um bom desempenho *na média*, para um caso genérico do problema. Escolher uma solução genérica implica em sacrifícios de desempenho em casos incomuns (Tiwari *et al.* (2011)).

O processo de otimização consiste em métodos de se escolher a melhor maneira de resolver um problema. Escolher este método é outro problema a ser resolvido, o que requisita esforço extra por parte do desenvolvedor. Planejar, testar, comparar e escolher os métodos a serem usados são tarefas que levam um tempo considerável de desenvolvimento para cada método. Quando existe ainda a implementação de mais de um método, usualmente a escolha entre eles fica por conta de alterar parâmetros do programa, que podem ser muitas vezes valores fixados no desenvolvimento, como constantes ou símbolos no código.

Hoos diz, em seu trabalho Programming by Optimization (Programar por Otimização, Hoos (2012)), que em sua experiência na área de desenvolvimento de solucionadores heurísticos de alto desempenho para diversos problemas combinatórios difíceis, construir software otimizado manualmente pelo desenvolvedor leva a resultados sub-otimais em termos de desempenho. Esse resultado mencionado se deve a tentativas de otimizar manualmente algoritmos complexos de maneira a fixar um método de resolução de um problema, devido a escolhas de design feitas pelo desenvolvedor baseadas em intuição, experiência prévia ou

alguma experimentação. O que então é sugerido no artigo de Hoos, é o conceito de PbO, que consiste em desenvolver diversas soluções para um mesmo problema, e utilizar um algoritmo que escolha quais são os melhores usos de cada solução. Hoos descreve uma plataforma de PbO que realiza o trabalho deste algoritmo de escolha para o desenvolvedor. Dados um problema, um código contendo diversas maneiras de se resolver este problema, e um *weaver* de PbO, é realizado pela plataforma então, um algoritmo que os trata como um problema de otimização estocástica. O *weaver* é responsável por tomar as partes do código a serem otimizadas e os trechos que contém as otimizações, e reuni-los em um único código coeso, funcional e otimizado. Esta otimização meta-algorítmica recebe um espaço de design, e um conjunto de entrada, e gera um solucionador que apresenta a melhor combinação de parâmetros que leva a um melhor desempenho. Esse método gera um código estático que possui as características desejadas, mas apenas para os casos relacionados. Os conceitos e métodos apresentados aqui aproximam-se muito de *autotuning*, na forma que ambos levam à uma otimização algorítmica de um programa.

Combinando as ideias de Programming by Optimization e *autotuning*, existe o PetaBricks (Ansel *et al.* (2009)), que é uma combinação de linguagem implicitamente paralela e de seu compilador, onde ter múltiplas implementações de múltiplos algoritmos para resolver um problema é a maneira natural de programar. O compilador do PetaBricks realiza o trabalho de *autotuning* sobre o código, fazendo tanto escolhas algorítmicas quanto escolhas de ajuste fino sobre os parâmetros do programa. O compilador também é responsável por fazer escolhas de técnicas automáticas de paralelização, distribuições de dados, parâmetros algorítmicos e transformações, entre outros.

Autotuning vem sendo usado em áreas como computação de alto desempenho e computação gráfica. Foi mostrado que consegue-se um desempenho melhor, ou pelo menos mais portável, utilizando-se *autotuning* comparado a desenvolver sem o uso da mesma (Ansel *et al.* (2014)). Mas, ainda assim, existem problemas e desafios relacionados com a aplicação de *autotuning* a projetos. Primeiramente, existe o problema de que *autotuners* são normalmente desenvolvidos de maneira específica para uma aplicação, de maneira que não podem ser usados com aplicações diferentes; também por este motivo, *autotuners* são trabalhosos de se implementar, por necessitar não apenas o desenvolvimento da aplicação base, mas um estudo de seus diferentes possíveis algoritmos, e de seu espaço de configurações. Esse esforço é muitas vezes considerado proibitivo em tempo de desenvolvimento.

Os desafios que são encontrados no desenvolvimento de *frameworks* de *autotuning* são, principalmente (Ansel *et al.* (2014)): encontrar a melhor representação de parâmetros para o programa; o tamanho do espaço de configurações válido, que pode ser proibitivamente grande (podendo passar de 10^{30} no nosso exemplo, mais detalhes na seção de Experimentos); e a complexidade topográfica deste espaço de configurações. Configurações podem ser representadas por diversos meios diferentes, desde simples valores numéricos, a listas e árvores de escolha representando um conjunto de instruções. Fica sob a responsabilidade do desenvolvedor escolher uma representação do problema que faça sentido para o *autotuner* de maneira que o problema seja tratável, o que altera a lógica do uso do *tuner*. Depois de se fixar uma representação, deve-se haver uma preocupação com o tamanho do espaço de configurações, pois ele pode ser demasiadamente grande, atingindo números de possibilidade que podem ser facilmente maiores que a quantidade de átomos do universo, que é da ordem de grandeza de 10^{79} [1].

Não apenas extensos, espaços de configurações são geralmente complexos, contendo ao mesmo tempo alta dimensionalidade, seções de crescimento ou decrescimento estáveis, platôs de continuidade, grandes espaços descontínuos, áreas onde certos parâmetros são altamente

[1] http://www.madsci.org/posts/archives/oct98/905633072.As.r.html, visitado em 29/11/15

relacionados e parâmetros completamente independentes entre si, dentre muitas outras características. Diferentes algoritmos de navegação destes espaços trabalham melhor com características diversas no espaço de busca, e se dedicar exclusivamente à uma ou poucas delas pode gerar ótimos locais, ou uma falha em obter-se uma otimização que efetivamente melhore o desempenho do programa.

Para tratar destes desafios então, foi desenvolvido um *framework* de *autotuning*, o OpenTuner (Ansel *et al.* (2014)). O OpenTuner permite a construção de *autotuners* específicos de aplicação de forma genérica, utilizando-se das suas próprias técnicas de busca para realizar a otimização via parâmetros da aplicação, desde que esta tenha alguma interface com o OpenTuner. Assim, qualquer problema que possa ter sua entrada representada por parâmetros e uma saída quantificável pode ser otimizado com este *framework*. Ele define tipos de dados e técnicas de busca, que tornam fácil preparar um novo projeto.

2.2 O OpenTuner

OpenTuner é um arcabouço para a implementação de sistemas de otimização e ajuste fino de programas. Utilizando um conjunto de técnicas empíricas de busca, o OpenTuner gera e testa combinações de parâmetros de configuração para um determinado programa, que podem representar por exemplo, escolhas algorítmicas.

OpenTuner introduziu o conceito de utilizar conjuntos de técnicas de busca para *autotuning* de programas, o que permite que um número de técnicas diferentes trabalhem em conjunto para encontrar uma solução ótima. Estes conjuntos incluem técnicas refinadas de busca, preparadas para navegar espaços complexos, e utilizar representações mais sofisticadas de dados; isto permite que problemas de complexidade maior sejam resolvidos, de uma maneira que pode ser adaptada por diversos projetos. O OpenTuner é capaz de encontrar resultados em espaços de busca muito grandes, passando de 10^{3600} possibilidades, como relatado no artigo (Ansel *et al.* (2014)).

O OpenTuner trata o problema de *autotuning* como um problema de busca. O espaço de busca é composto de configurações, que são atribuições concretas de um conjunto de parâmetros. Parâmetros podem assumir diversas formas, como números simples, ou conjuntos complexos de dados. Um resultado é gerado executando-se o conjunto de configurações em uma forma específica ao seu domínio, e pode ser medido em desempenho, corretude da resposta ou outras métricas. Técnicas de busca são responsáveis por alterar as configurações utilizando manipuladores definidos pelo usuário. A figura 2.1 ilustra a organização do funcionamento do OpenTuner.

O sistema de conjuntos de busca do OpenTuner funciona agregando diversas técnicas de busca em uma metatécnica, que delega a busca a uma delas. Dessa forma, é possível montar uma hierarquia de técnicas e metatécnicas que permita a melhor solução para cada problema. O mecanismo de busca base do OpenTuner age em cima apenas de uma técnica, geralmente uma metatécnica. A metatécnica central do OpenTuner é a *multi-armed bandit with sliding window, area under the curve credit assignment (AUC Bandit)*, baseada numa solução ótima para o problema de múltiplos caça-níqueis Fialho *et al.* (2010). Esse problema consiste em encontrar um equilíbrio entre, dadas múltiplas máquinas caça-níquel, explorar quais são os que oferecem melhores recompensas e utilizar com mais frequência os que possuem resultados melhores. Para este trabalho, foram utilizados os métodos de busca padrão do OpenTuner.

Um exemplo de aplicação do OpenTuner, mostrado no artigo Ansel *et al.* (2014) é o de escolha de *flags* de compilação na execução do gcc, o compilador padrão de C. Foram usados diferentes programas a serem otimizados, e o OpenTuner determinava o conjunto de *flags* que produziria um binário executável com o melhor tempo de execução. O artigo mostra

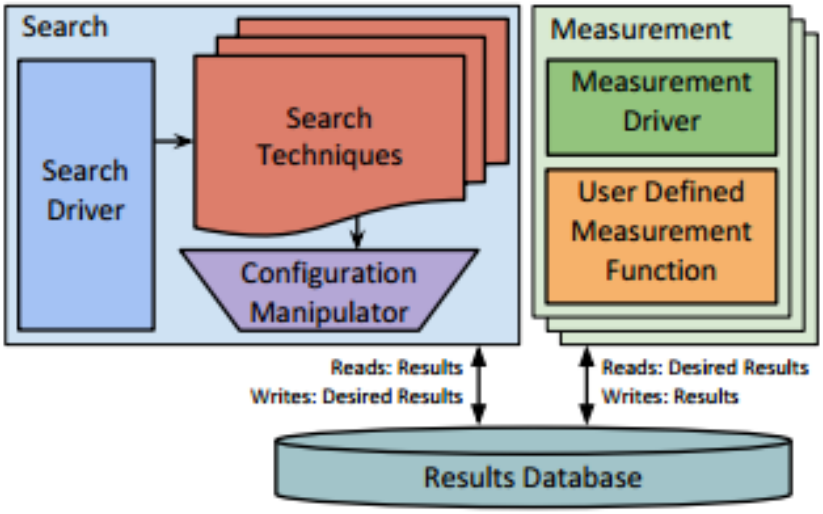

Figura 2.1: *Diagrama de Funcionamento do OpenTuner*
Fonte: Ansel et al. (2014)

os resultados, e com o OpenTuner foi obtido um *speedup* de até 2.8 vezes sobre a *flag* de otimização básica -O3 do gcc.

Decidimos por estudar aplicações do OpenTuner no mundo real, em específico para jogos digitais. A ideia básica é que podemos usar o OpenTuner para otimizar um conjunto de parâmetros dentro do jogo, para ajudar o jogador a ter um planejamento, ou como uma metaferramenta de auxílio. Exemplos são a otimização de: parâmetros de inteligências artificiais para obter comportamentos desejáveis; parâmetros de geradores de mapas para obter ambientes variados com características comuns desejáveis; e a distribuição de pontos em personagens de um RPG (Role-Playing Game).

2.3 Estudos Preliminares

Inicialmente, era desejado a utilização do OpenTuner em otimização de parâmetros em geradores de mapas. Seu propósito seria ser uma metaferramenta sobre o gerador de mapas de algum jogo, para otimizar a geração aleatória ou determinística de mapas com características desejáveis e quantificáveis.

Várias ideias de diferentes jogos possíveis foram elencadas, e inicialmente foi decidida a utilização de um gerador de mapas para o jogo DooM. Sendo um jogo consideravelmente antigo e simples, o trabalho seria também simples. Não só existem implementações *open-source*, como foi encontrado um gerador de mapas de simples uso que poderia ser usado em *autotuning*.

O fato de DooM ser um jogo muito antigo foi desfavorável ao projeto. Uma boa parte da comunidade de desenvolvedores de conteúdo sobre o jogo é extremamente antiga, com seus sites e fóruns desativados ou grandemente desatualizados. Mas o maior problema era de ordem conceitual: como quantificar a qualidade de um mapa? Para o processo de *autotuning*, é necessário um resultado numérico, que não se pode derivar facilmente de um mapa, sem técnicas mais complexas; e mesmo que um valor fosse derivado, não havia como garantir que era uma representação acurada da *fitness* do mapa. A *fitness* do mapa seria um valor ou conjunto de valores extraídos algoritmicamente do mapa, de forma que fosse possível que outros algoritmos o julgassem como um mapa "bom"ou "ruim"de acordo com as regras

colocadas pelo desenvolvedor.

Foi considerada a ideia de usar um *bot*, um personagem controlado por IA, como função de *fitness* para o mapa, como maneira de quantificar a qualidade do mapa, mas informações sobre o funcionamento do jogo são escassas. O plano inicial seria realizar *autotuning* no gerador de mapas, e como uma maneira de "otimização dupla"utilizar um *bot* pré-otimizado para avaliar se o mapa gerado contém as características necessárias. Quantidade de inimigos, tempo decorrido até o mapa ser completado pelo *bot*, quantidade de recursos gastos, seriam os valores avaliados pelo *autotuner*. O tamanho da tarefa de encontrar um jogo, um gerador de mapas e um conjunto de *bot* com os quais se pudesse trabalhar provou-se grande demais para o escopo deste trabalho.

Próximo na lista de ideias estava a otimização da distribuição de pontos para um personagem de RPG (Role-Playing Game, ou jogo de interpretação de papeis). Para cada personagem em um jogo de RPG, existe um conjunto de atributos, com pontos que podem ser distribuídos entre eles; para cada atributo, uma determinada quantidade de pontos afeta de forma diferente os resultados finais que um personagem pode ter no decorrer do jogo (um maior valor de Força, por exemplo, acarreta em ataques mais poderosos). Computacionalmente é uma questão muito mais simples, pois com esse exemplo consegue-se derivar diretamente os valores a serem otimizados. Exemplos destes valores são: a quantidade de pontos de vida que um personagem possui ou quantos pontos de dano o mesmo causa com um ataque.

A escolha de qual jogo se usaria de base torna-se então o problema. Deve-se considerar a complexidade do jogo em si, dado que isto influencia diretamente a maneira de se modelar a distribuição de pontos no OpenTuner. Foi iniciada a implementação de um *autotuner* simples com uma árvore de habilidades genérica como exemplo, para verificação de como o programa se comportaria, e se ele daria os resultados ótimos dentro de um bom tempo de execução.

Uma árvore de habilidades em um jogo de RPG, como mencionado no parágrafo anterior, é um dos aspectos do jogo a ser decidido pelo jogador, de forma a se customizar seu personagem. A árvore contém nós que representam diferentes habilidades que influenciam o desempenho do tal personagem dentro do jogo, modificando que coisas ele pode fazer, ou quão bem essas coisas serão feitas (ataques mais fortes, como o exemplo mais básico disso). É possível derivar-se então que o desejado seria encontrar uma maneira ótima de se preencher essa árvore, pois escolher alocar seus limitados pontos em uma maneira implica em focar alguns aspectos da árvore, e sacrificar outros. Essa prática é conhecida entre jogadores pelo termo *minmaxing*[2].

Ideias de jogos a serem modelados foram: Torchlight 2[3], Path of Exile[4], The Elder Scrolls V: Skyrim[5], jogos do tipo Rogue-like[6], League of Legends[7], DOTA 2[8], e até algum possível sistema de RPG de mesa. Alguns destes foram descartados imediatamente por motivos como dificuldade de modelagem da árvore ou aleatoriedade demasiada envolvida no processo de testes.

Foi considerada então a possibilidade de trabalhar com os jogos Torchlight 2 ou The Elder Scrolls V: Skyrim. Esses dois jogos possuem árvores de habilidades razoavelmente simples conceitualmente; então era necessário verificar a viabilidade da interação dos mesmos com o Python, para fazer-se a interface com o OpenTuner. Novos problemas foram

[2] http://rpg.stackexchange.com/questions/64800/what-does-minmax-mean, visitado em 20/01/16
[3] http://www.torchlight2game.com/, visitado em 29/11/15
[4] https://www.pathofexile.com/, visitado em 29/11/15
[5] http://www.elderscrolls.com/skyrim/, visitado em 29/11/15
[6] https://pt.wikipedia.org/wiki/Roguelike, visitado em 29/11/15
[7] http://br.leagueoflegends.com/, visitado em 29/11/15
[8] http://br.dota2.com/, visitado em 29/11/15

encontrados, na forma de incompatibilidades técnicas. As comunidades de desenvolvimento de conteúdo de Torchlight e de Skyrim trabalham primariamente com ferramentas fornecidas pelos desenvolvedores, que envolvem plataformas e linguagens de *scripting* próprias dos jogos, sem interação com linha de comando externa, ou possibilidade de alteração de atributos de personagem por ferramentas externas ao jogo. Em ambos os jogos, de maneiras similares, apenas é possível manipular o jogo com uma ferramenta externa própria, feita pelos desenvolvedores, ou internamente por um console isolado do sistema operacional, com comandos limitados. Adicionalmente, tentativas de alterar os arquivos de dados de personagem dos jogos mostraram-se demasiadamente complexas, por envolverem arquivos encriptados, e não existir nenhum método diretamente acessível de realizar esta decodificação.

Uma ideia que foi sugerida à parte destas outras, apenas pela discussão dos métodos, foi criar um *bot* que apostasse no site SaltyBet. Esse site consiste de uma *stream* de vídeo com um jogo de luta sem jogadores, apenas com IAs controlando os personagens. Usuários que acessam o site podem apostar dinheiro virtual, referido como "Salt Dollars", no resultado da luta. O *bot* leria as entradas do chat, onde os personagens envolvidos e os resultados são anunciados, e daria estas informações para o OpenTuner. O OpenTuner, por sua vez, gravaria os resultados obtidos, e escolheria a *melhor estratégia de apostas* para decidir em qual personagem se apostaria, e quanto. Sendo apenas uma discussão de viabilidade, esta ideia não foi seguida.

Finalmente, o jogo que foi escolhido para este estudo de viabilidade foi o OpenTTD. É um simulador de gerência de empresa de transportes, o que o distancia dos gêneros dos jogos estudados anteriormente. O que realmente causou a sua escolha para o projeto foram o fato de ser um jogo de código aberto, e a existência de uma simples forma de interface com a linha de comando do sistema.

Capítulo 3

OpenTTD

3.1 O Jogo

O OpenTTD [1] é um clone de código aberto do clássico jogo Transport Tycoon Deluxe. Neste jogo, o jogador assume o papel de um administrador de uma empresa de transportes, e pode gerenciar e construir diversas redes de transporte, desde operar linhas de ônibus, até construir ferrovias e aeroportos. O jogo não possui objetivo específico ditado para o jogador, deixando ao mesmo a liberdade de escolher qual será. Pode-se querer construir uma grande linha de trem, preparar pontos de ônibus, ou apenas jogar o jogo com o desejo de ver a empresa virtual crescer. Pela liberdade de objetivo e pela simplicidade de lidar com o jogo, ele foi escolhido para ser tratado neste trabalho.

Figura 3.1: *Tela Inicial do OpenTTD*

OpenTTD foi baseado no jogo original de 1994, que recebeu um *patch* feito pela comunidade em 1996. A versão de código aberto foi lançada como uma derivação de tal patch chamado TTDPatch, que visava resolver alguns problemas de ordem técnica do jogo, além

[1] www.openttd.org

de adicionar algumas funcionalidades como novos gráficos, veículos e indústrias. Entretanto, esse *patch* não podia alterar facetas mais profundas do jogo original, e era limitado aos sistemas e plataformas nos quais este poderia ser executado. Seguindo o mesmo espírito deste *patch*, o OpenTTD foi criado como uma obra de engenharia reversa do jogo original, recriando-o em linguagem C. O projeto começou em 2003, por autoria de Ludvig Strigeus, que desejava uma maneira de continuar jogando Transport Tycoon Deluxe que se adaptasse a novas tecnologias, e pudesse ser modificado de maneiras mais profundas pela comunidade. O jogo foi lançado oficialmente em 2004, e até o ano de 2010, ainda era dependente dos gráficos, músicas e efeitos sonoros do jogo original. Estes recursos foram gradualmente substituídos por versões feitas pela comunidade, tornando-o independente dos materiais do jogo original.[2]

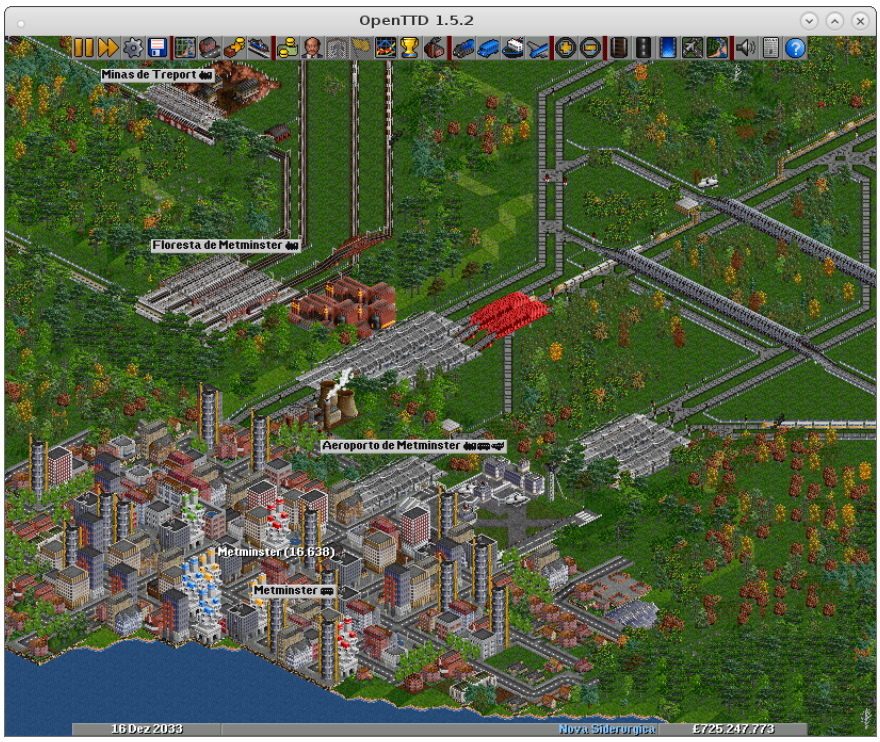

Figura 3.2: *Screenshot de gameplay do OpenTTD*

No jogo, a única forma de se ganhar dinheiro é com o transporte de produtos ou passageiros de um lugar de origem, que produz esses recursos, até um local de destino, que os consome. Cada local produtor produz recursos diferentes, como uma fazenda que produz grãos e gado, e cada local consumidor consome um recurso diferente, como uma fábrica que consome aço, grãos ou gado. Alguns consumidores, por sua vez, geram outros produtos ao consumirem, como as já mencionadas fábricas, que ao receberem recursos, produzem bens.

Para cada tipo de recurso, porém, não há discriminação além de que tipo é produzido ou consumido. Isso significa, por exemplo, que passageiros embarcam em qualquer estação e sempre desembarcam na próxima, não tendo um destino específico. O dinheiro recebido a cada transporte depende do tempo entre a produção e o consumo, e a distância entre o produtor e o consumidor. Quanto maior a demora, menor o pagamento, e quanto maior a distância, maior o dinheiro recebido. A carga e descarga é realizada em estações dedicadas, que devem estar próximas dos pontos de interesse. É possível juntar estações de veículos diferentes para serem tratados como uma única estação, compartilhando recursos que podem ser transportados por cada um dos veículos que a utilizam.

[2] https://www.openttd.org/en/about, visitado em 29/11/15

Além disso, há diversos meios de transporte, cada um com suas características. Trens, por exemplo, são caros, precisam de um vagão dedicado à locomoção e vagões separados para carga, e circulam apenas em ferrovias, mas são rápidos e têm uma quantidade ajustável de carga que pode ser transportada, de acordo com o número de vagões dedicados a cada tipo de produto. Caminhões e ônibus, por outro lado, são baratos e capazes de carregar produtos sem mais auxílio, mas são lentos e tem pouca capacidade. Há também aviões e navios que podem ser utilizados. Existem diversos modelos diferentes para cada uma dessas categorias. A seleção de veículos muda conforme o tempo do jogo progride, com novos modelos tornando-se disponíveis e os antiquados não podendo mais ser adquiridos.

O OpenTTD tem suporte a execução como servidor dedicado, sem interface gráfica. Nesse modo, a interação com ele se dá através de um console, que aceita comandos pré-determinados, como iniciar uma IA ou reiniciar o jogo, e escreve na saída padrão e na saída de erro os acontecimentos do jogo.

Uma sessão de jogo normal começa com a criação do mapa do jogo. Quando o jogador clica no botão "New Game", a interface de geração de mapa é aberta; nela, pode-se escolher os parâmetros do algoritmo de criação de mapa, como o tamanho, o tipo de terreno e a densidade de cidades. Um outro detalhe importante é a possibilidade de se escolher a semente do gerador de números aleatórios a ser usada para este mapa. Quando o jogador estiver satisfeito com as escolhas neste menu, o mapa e criado e o jogo é iniciado. Por padrão, o jogo inicia no ano de 1950, e o jogador é responsável por tentar fazer a sua empresa de transportes crescer. A empresa possui uma quantidade inicial de fundos para começar suas operações, e é possível pedir empréstimos para o banco, que deverão ser pagos no futuro. O jogo termina se a empresa não tem dinheiro para continuar operando nem pagar os empréstimos, declarando falência.

Para se começar a desenvolver a empresa no jogo, o jogador precisa primeiro identificar qual será o tipo de transporte a ser feito. Começando com o tipo mais simples e barato, o transporte de passageiros via ônibus, o jogador precisa construir pelo menos uma garagem de veículos e duas estações para formar um trajeto; preferencialmente estes serão construídos em uma cidade com pelo menos 500 habitantes, e com alguma distância entre eles. Com esta infra-estrutura construída, aproveitando-se as ruas da cidade, o jogador pode comprar um ônibus, e dar ordens para ele, determinando que sua rota será entre as estações. Ao chegar no final de sua lista de instruções, um veículo sempre irá repeti-las do começo. Conforme o veículo percorre seu trajeto, ele começa a gerar dinheiro para a empresa conforme o tipo de carga e a distância percorrida, além do tempo levado para percorrer esta distância. Assim sendo, muitas vezes trens são considerados o tipo de transporte mais rentável, por carregar grandes quantidades de carga e passageiros a longas distâncias. Trens são balanceados por possuírem o maior custo inicial de preparação da linha, necessitando da construção de ferrovias e estações de trem em cada ponto desejado da rota. Para transportes de longas distâncias, em que o tempo de entrega é importante para manter a taxa de lucro, são usados aeroportos e transportes aéreos, que possuem os maiores custos de operação, mas geram a maior quantidade de dinheiro por viagem.

Construindo diferentes rotas de transporte e realizando a manutenção e melhoramentos em sua frota de veículos, o jogador segue então pelo jogo expandindo sua empresa e acumulando dinheiro. O calendário do jogo vai até o ano 2050, quando não existem mais melhoramentos disponíveis para os veículos do jogo. Apesar disso, o jogador pode continuar jogando, apenas com um nível agora fixo de tecnologia para os veículos. Todas as ações disponíveis ao jogador até este momento continuam possíveis.

3.2 Inteligências Artificiais

O OpenTTD tem suporte a inteligências artificiais, cujo comportamento é definido por um conjunto de *scripts* Squirrel. A linguagem Squirrel é descrita pelos seus desenvolvedores como uma linguagem de *scripting* de alto nível, imperativa e orientada a objetos. Projetada para ser leve e se adequar aos requerimentos de banda, memória e processamento de aplicações como jogos[3], sua sintaxe é bastante similar à de C++. O OpenTTD tem seu próprio interpretador Squirrel, que executa um certo número de instruções de cada IA antes de interrompê-la por um breve período de tempo.

Esses *scripts* são responsáveis por todas as ações que a IA pode tomar, como quais pontos de produção e consumo atender com seus veículos, como construir um caminho que atenda a este transporte, e quais veículos utilizar para isso. Essas decisões são tomadas no mesmo nível que um jogador humano normalmente tomaria. Ou seja, são decisões em nível de gerenciamento de empresa, como: planejamento e construção de vias a serem utilizadas pelos veículos, pagamento ou aquisição de empréstimos; mas não incluindo controle fino sobre a operação dos veículos. Os mecanismos de funcionamento da IA podem ter uma variedade muito grande de escopo e complexidade, desde IAs simples que focam apenas em um tipo de veículo, com todo o código contido em apenas um arquivo de *script*, até IAs que precisam de diversos arquivos e bibliotecas auxiliares em seu funcionamento. Existem também IAs cujo foco não é o desempenho da empresa, tendo como objetivo o plantio de árvores, simulação de tráfego local ou gerenciamento básico de veículos para jogadores que não quiserem fazer isso manualmente.

Cada conjunto de *scripts* fica dentro de uma pasta, que fica em um diretório pertencente ao de configurações do OpenTTD. Esse diretório de IAs também pode incluir bibliotecas para o uso de outras IAs. Cada IA deve ter ao menos um *script* de informações de identificação da IA e um *script* principal que deve conter um *loop* infinito com seu funcionamento. Cada IA pode ter também qualquer número de *scripts* auxiliares, que devem ficar na mesma pasta e serem chamados a partir do *script* principal. Essas IAs podem ser instanciadas durante o jogo, e cada uma que seja iniciada passa a assumir o controle de uma companhia, cujo comportamento é definido pelos seus *scripts*.

[3] www.squirrel-lang.org, visitado em 29/11/15

Capítulo 4

Implementação

4.1 Objetivos

Para avaliar a aplicabilidade de técnicas de ajuste fino em jogos, neste trabalho foi feito um *tuner* para o vetor de custos de *Pathfinding* para construção de trilhos da IA, além do custo máximo admitido pelo algoritmo, por ser um conjunto de valores mais simples de se trabalhar. Cada posição desse vetor representa um custo associado a um tipo de trilho construído sob diferentes condições, como curvas, pontes, subidas ou descidas, sendo que o caminho escolhido será o de menor custo que seja capaz de conectar a origem ao destino. Note que esse custo não é diretamente relacionado à quantia de dinheiro gasta pela IA no jogo para construir o caminho. Serão observados fatores como facilidade de implementação, funcionamento adequado e resultados obtidos. A função a ser otimizada é diferente em cada um dos testes, sempre tentando maximizar algum valor ou recurso em determinados períodos de tempo de jogo. Esses testes serão detalhados mais à frente. Para este trabalho, foram utilizados os métodos de busca padrão do OpenTuner.

4.2 Integração

O *autotuner* implementado (Figura 4.1) é composto pelo módulos TTDTuner, TTDHandler e AIBuilder. O TTDTuner interage com o OpenTuner, TTDHandler interage com o OpenTTD, e o AIBuilder constrói as IAs. Além destes, uma IA base que é uma versão modificada e incompleta dos arquivos da IA ChooChoo[1]. Essa IA, além de ter os custos de *pathfinding* removidos para serem determinados pelo *autotuner*, tem mensagens de saída adicionadas para relatar os resultados depois de um certo tempo. Existe um arquivo de configuração, o qual determina a pasta onde está a IA base, onde fica o diretório de IAs do OpenTTD, e qual o comando a ser utilizado para iniciar o OpenTTD. Por linha de comando, pode-se especificar qual valor será observado, quantos anos irá durar cada iteração da simulação, e quantas IAs existirão por iteração. Os valores medidos podem ser valor da empresa, lucro no último quartil, ou dinheiro em caixa.

O TTDTuner recebe as configurações do OpenTuner, escolhe um número de ID e inicia o AIBuilder, o qual constrói uma IA e a coloca na pasta apropriada do OpenTTD, e devolve o nome da IA gerada. Em seguida, o TTDTuner repassa esse nome para o TTDHandler, que carrega estas IAs no jogo. O TTDTuner então aguarda o final da iteração, marcado pelo recebimento dos resultados vindos do TTDHandler, e então requisita que o mesmo reinicie a sessão de jogo. Por final, a média dos resultados é tirada e enviada para o OpenTuner.

[1] http://www.tt-forums.net/viewtopic.php?t=44225, visitado em 29/11/15

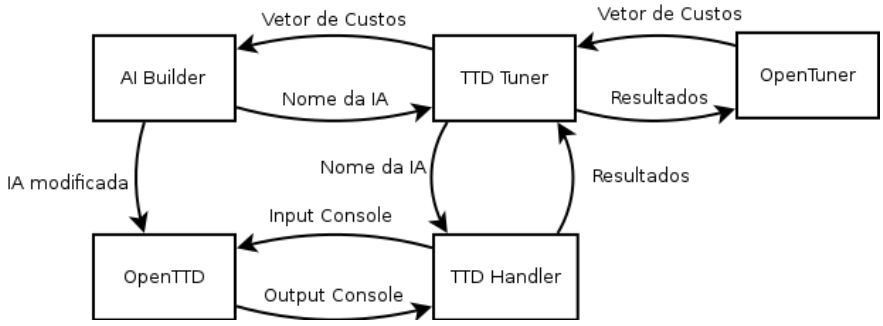

Figura 4.1: *Estrutura do tuner implementado*

O AIBuilder funciona substituindo linhas com palavras-chave pré-determinadas em cada um dos arquivos a serem modificados, com cada palavra-chave sendo mapeada a uma substituição diferente. A lista de palavras-chave e o mapeamento destas é independente para cada arquivo. O AIBuilder recebe um vetor de custos, um nome de parâmetro a ser avaliado, um número de anos e um ID do TTDTuner, que são usados para construir adequadamente as instruções que serão inseridas no código da IA gerada. O nome da IA é determinado pelo ID recebido, e é passado de volta para o TTDTuner.

O TTDHandler realiza a interface com o OpenTTD. Ele envia comandos para o servidor via pipe para o stdin do servidor e lê a saída retornada. O TTDHandler é responsável por iniciar IAs, parar IAs, ler respostas, e reiniciar o servidor quando necessário. Após iniciar as IAs, ele espera uma saída em formato específico vinda do jogo, formato este definido pelo padrão do OpenTTD e por uma convenção adotada para o ajuste fino realizado aqui. Ao detectar essa saída, que contém o resultado obtido e a identificação da IA, o TTDHandler concatena esse resultado em um vetor, e devolve esse vetor quando todas as IAs terminarem.

4.3 Experimentos e Resultados

Para verificar o funcionamento do *autotuner*, foram realizados testes de 10 e 50 anos com 8 IAs por iteração para valor da empresa, e de 10 anos com 6 IAs e 30 anos com 8 IAs para dinheiro em caixa e lucro. Testes de 10 anos foram executados numa máquina virtual cedida por William Gnann, com 512MB RAM e um núcleo de um processador AMD Opteron 2210, e os testes mais longos foram feitos na Rá, o servidor web do USPGameDev, com 4GB de RAM e um processador Intel Xeon X3430. Ambas estavam com o sistema operacional Debian 7.

Os testes consistem de uma execução do *autotuner*, onde, a cada rodada, é escolhida uma configuração para o vetor de custos do *pathfinder* da IA, o jogo instancia o número determinados de cópias dessa IA, os resultados observados são coletados e informados de volta para o OpenTuner, que escolhe uma nova configuração para a próxima iteração com base nos resultados observados.

Para a realização dos experimentos relacionados neste trabalho, o OpenTTD foi compilado com uma alteração de uma *flag* que permite que os ciclos de jogo passem de forma muito mais rápida que o natural. Assim, a passagem de tempo dentro do jogo é acelerada de modo a possibilitar a geração de mais resultados em menos tempo. Os testes realizam rodadas de 10, 30 e 50 anos dentro do jogo, sendo que um ano de jogo dura aproximadamente 13 minutos. Realizar baterias de testes levando 10 horas cada uma seria proibitivo.

A intenção original seria rodar esses testes por uma semana, mas instabilidades causando que os mesmos fossem interrompidos ocorriam em pontos imprevisíveis da execução, entre

8 e 72 horas após o início dos testes. Para os testes de caixa e lucro, o valor de semente inicial para o gerador de mapa do OpenTTD foi fixado, mas não durante os testes de valor da empresa. Esta alteração foi pensada depois dos testes iniciais, como uma maneira de eliminar a influência de aleatoriedade do ambiente do jogo sobre os resultados.

Uma observação à parte sobre os testes realizados: em determinado momento durante os testes no servidor Rá, foi tentada uma conexão externa com as instâncias de teste do jogo. O próprio sistema do jogo recusou a conexão, por assumir que o cliente estava rodando em uma velocidade lenta demais para a conexão. Isto aconteceu devido à compilação para alta velocidade feita para o OpenTTD.

A seguir, estão os gráficos criados com os resultados obtidos nos experimentos. Neste conjunto de gráficos, o gráfico à esquerda mostra os valores observados nas medições, e os gráficos à direita mostram apenas o máximo valor encontrado, para facilidade de observação da melhora dos resultados. A linha verde indica a média dos valores obtidos com a IA original, como forma de comparação de desempenho.

Pode-se verificar, no geral, que o OpenTuner consegue obter resultados melhores conforme mais iterações passam, sempre tendo o melhor valor final significantemente maior que os valores obtidos pela IA original. Como os gráficos do valor máximo tem seu eixos verticais escalados para melhor mostrar a variância dos máximos locais, a média original encontra-se perto da base do gráfico, ou até mesmo fora dele, em alguns casos. Na figura 4.2, que mostra um teste mais longo em tempo do jogo, podemos ver que as iterações iniciais mantiveram-se em uma média, mas valores melhores foram encontrados em iterações mais avançadas, como era de se esperar. Pode-se observar como o valor original foi rapidamente superado, e o *autotuner* continuou a melhorar seus resultados. Este teste ainda continha uma certa variância estocástica por interação do gerador de mapas com os testes; a semente do gerador não havia ainda sido fixada, o que causou as oscilações de valor observadas.

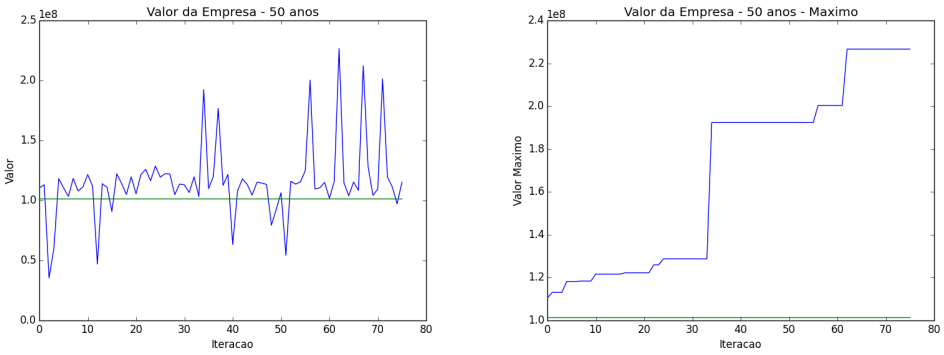

Figura 4.2: *Experimento de valor de 50 anos*

Na figura 4.3 pode-se ver os resultados de um teste similar, ainda utilizando o valor da empresa como resultado observado. Este teste, entretanto, tem uma duração de 10 anos do jogo. Um teste mais curto possibilita uma melhor coleta de dados, pois podem-se executar mais iterações em menos tempo. Com mais iterações, é possível observar os valores do teste oscilando mais acima e abaixo dos valores originais; porém, os melhores valores observados rapidamente os superam, novamente com uma diferença significativa. Outra diferença com relação ao teste anterior é o fato de o gerador de mapas ter sua semente fixada, removendo o fator aleatório do mapa do teste, possibilitando resultados mais acurados. A redução da duração foi feita como uma maneira de se obter mais resultados, pois iterações de 50 anos levavam muito tempo para serem completas, como explicado anteriormente.

Nas figuras 4.4 e 4.5 podemos ver testes realizados utilizando uma forma diferente de

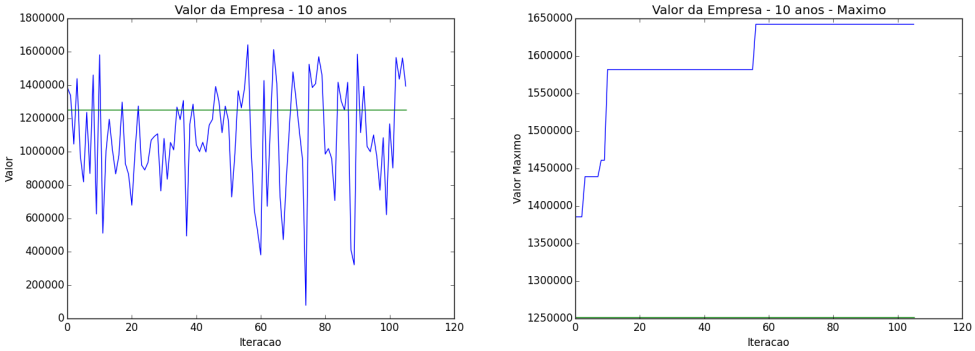

Figura 4.3: *Experimento de valor de 10 anos*

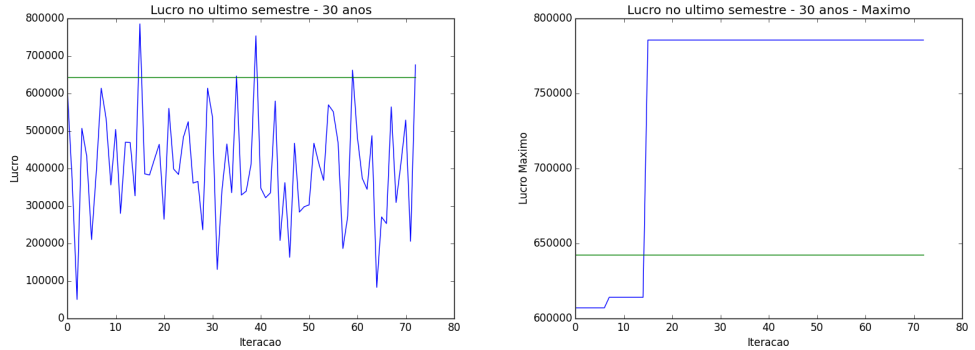

Figura 4.4: *Experimento de lucro de 30 anos*

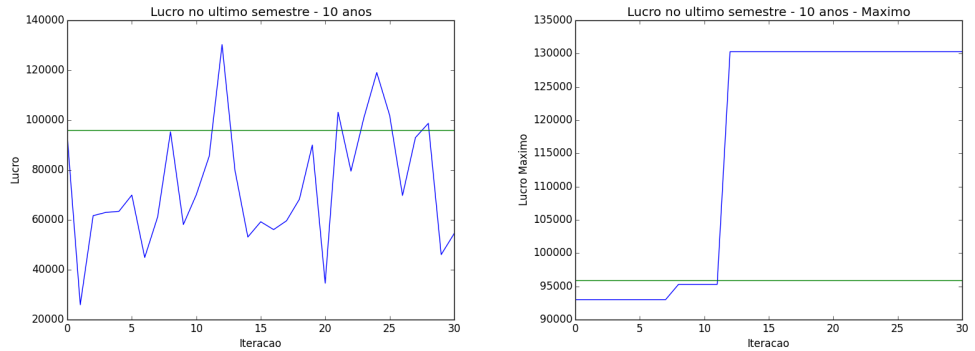

Figura 4.5: *Experimento de lucro de 10 anos*

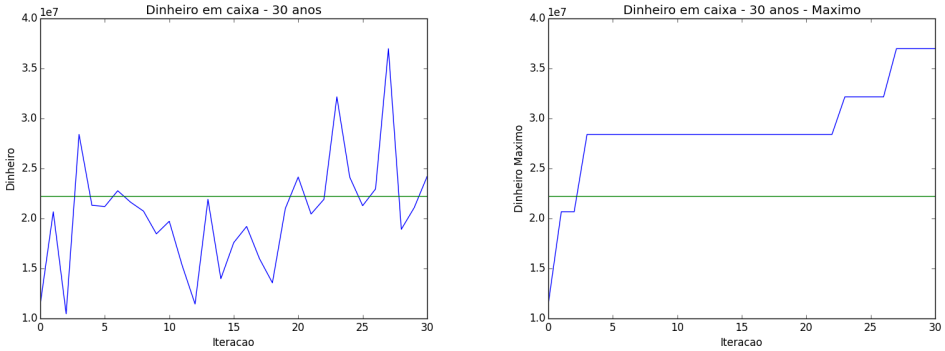

Figura 4.6: *Experimento de caixa de 30 anos*

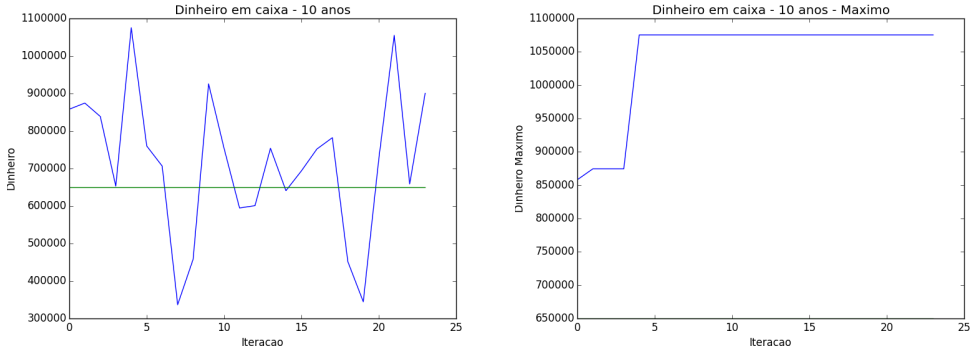

Figura 4.7: *Experimento de caixa de 10 anos*

verificar o resultado. Também sendo duas variações diferentes de tempo de execução devido à duração dos testes em tempo de jogo, consegue-se uma melhor observação da influência dos testes nos resultados. Nesses testes, o lucro da empresa ao final de cada iteração foi medido, como uma alternativa ao valor da empresa. Utilizando o lucro, pode-se ter uma visão melhor do crescimento da empresa sobre os anos, e pode-se observar a melhora encontrada pelas técnicas de busca do OpenTuner. Neste teste pode-se observar, comparando os valores encontrados com a média dos valores originais, que o *autotuner* possui uma oscilação de valores que costuma obter resultados abaixo da média, mas continua a encontrar melhores valores máximos, o que contribui para o objetivo.

Por fim, são mostrados os gráficos dos experimentos onde o valor em caixa ao final de cada quartil foi medido, tanto com testes de 30 anos em jogo quanto com testes com a duração mais baixa de 10 anos de jogo. Pode-se observar na figura 4.7 que, em casos em que não são executadas muitas iterações, o OpenTuner pode encontrar um melhor resultado nas primeiras iterações, e suas técnicas de busca não conseguem encontrar resultados melhores por algum tempo. Também pode-se ver que a linha da média dos valores originais ficou deslocada no eixo vertical, devido à diferença na escala dos valores. Com iterações suficientes, como pode-se observar na figura 4.6, é possível ver um crescimento estável do resultado, subindo gradativamente acima dos valores originais, mostrando que as técnicas de busca conseguiram funcionar corretamente.

Capítulo 5

Conclusões

Com este trabalho, foi explorada a viabilidade de aplicações do arcabouço OpenTuner em jogos digitais. Por extensão, é considerado também um estudo de aplicação do conceito de *autotuning* em geral a jogos. Para isso, foram consideradas diversas abordagens possíveis de aplicações em jogos, até que fosse encontrada a que foi explorada de fato. Encontrar um aspecto de um jogo já existente que pudesse ser otimizado com *autotuning* foi um desafio à parte.

Parte do propósito do estudo de viabilidade era encontrar esta combinação de um jogo e um aspecto a ser otimizado. A pesquisa de possíveis combinações levou à muitas ideias interessantes, dentre as quais foi escolhida a otimização de um aspecto das IAs do jogo OpenTTD. Assim que o jogo foi fixado, os esforços para integração mostraram que, em caso de se obter uma interface entre o jogo e o OpenTuner, pode-se otimizar um aspecto do jogo com sucesso.

Apesar de todos os empecilhos, foi observado que o OpenTuner conseguiu melhorar os resultados observados modificando apenas o vetor de custos do algoritmo de *Pathfinding* usado pela IA. Mesmo esta sendo uma parte relativamente pequena de uma IA bastante complexa, e sem saber o funcionamento interno do jogo ou da IA, pode-se ver a diferença nos resultados obtidos. No momento, esses resultados podem ser questionáveis, devido à forma como o OpenTTD lida com as IAs, já que as medições podem ter tido uma variância considerável de tempo do jogo entre as medições.

Com isso em mente, talvez seja possível obter resultados melhores se o jogo for desenvolvido com foco em automatização desde as etapas iniciais, e estruturando o *autotuner* de modo a aproveitar melhor as características do OpenTuner. A construção de um jogo tendo em mente seu uso com um *autotuner* desde as fases de concepção pode ser um esforço altamente interessante, pois não só os mecanismos internos e algoritmos do código do jogo podem ser otimizados via *autotuning*, como também aspectos de *gameplay* propriamente dito podem ser alterados e estudados utilizando um *autotuner*.

Capítulo 6

Parte Subjetiva

6.1 Renan Teruo Carneiro

6.1.1 Desafios e Frustrações

Tivemos dificuldades em achar um jogo bom para realizar o tuning. Queríamos evitar criar um jogo só para isso, especialmente dado que as ideias iniciais exigiriam muito esforço só para desenvolver a parte a ser otimizada, e isso sem contar a implementação do tuner em si. Porém, praticamente todos os jogos que chegamos a discutir tinha algum problema, especialmente para automatizar os testes necessários para o processo. Além disso, extrair os dados gerados e analisá-los externamente seria difícil, devido à forma como são guardados, como também exigiria um conhecimento mais profundo de mecânicas e jogabilidade do jogo para ter uma modelagem aceitável, derrotando de certa forma o propósito de usar o OpenTuner.

Além disso, houve dificuldades em automatizar o OpenTTD, que não pôde ser compilado se simultaneamente não houvesse interface gráfica e networking, exigia um conjunto de gráficos mesmo sendo executado em modo servidor dedicado, e compilava e carregava IAs pela primeira vez sem problemas sem suporte à biblioteca libzlo2, mas falhava ao reiniciar o jogo se fosse compilado dessa forma.

A falta de paralelismo na execução de testes do OpenTuner também foi um problema, já que no nosso caso, já que o jogo roda numa única thread, isso faria sentido. Seria possível usar uma parte do OpenTuner de forma meio errada para subverter isso, mas com seus próprios problemas. Outro problema decorrente disso é a estrutura do Handler, que foi inicialmente projetado para interagir com diversas threads, cada uma correspondente a cada uma das IAs que estariam todas rodando no mesmo servidor. Essa estrutura acabou ficando complexa demais para o funcionamento atual, e isso pode ter sido um dos fatores que ocasionaram as falhas na execução de testes.

Falando das falhas, essas foram um problema curioso. Não percebemos nenhum indício que aponta mais precisamente por que alguma coisa em algum momento parava de funcionar. Na Rá,simplesmente executava um número não fixo de iterações por um período de tempo não fixo e parava, sem mais mensagens, inclusive as de erros. Na máquina virtual, provavelmente acabava algum recurso eventualmente, já que acusava um erro de não adquirir um lock que era adquirido na linha imediatamente anterior. Ainda assim, não conseguimos identificar qual recurso estava vazando, e se isso de fato acontecia.

6.1.2 Disciplinas Relacionadas

Inteligência Artificial, já que estávamos alterando um vetor de custos de um A*.
Aprendizagem Computacional, que é basicamente a base de tudo isso.
Programação Concorrente para lidar com a estrutura que deveria ser multithreaded do tuner mas que acabou não sendo.
Redes teria sido útil se o OpenTuner fosse paralelizado, já que íamos fazer um tuner cujas threads se comunicariam com o handler através de conexões no localhost, mas isso nunca foi implementado.
Laboratório de Programação porque Git e Latex.
Laboratório de Programação II porque POO simples.

6.2 Vitor Cerqueira Santos

6.2.1 Desafios e Dificuldades

Tivemos diversos desafios durante o ano no desenvolvimento deste trabalho. A começar por pensar em qual aspecto de um jogo seria otimizado, porque isto envolvia escolher um jogo que pudesse ter este aspecto explorado. A busca por um jogo que fosse simultaneamente acessível tecnicamente e que possuísse um aspecto interessante a ser otimizado. Chegamos a considerar criar um jogo simples para ser otimizado, mas isto geraria uma necessidade muito maior de esforço, que não seria viável para este trabalho.

Na pesquisa necessária para escolher o jogo a ser trabalhado, diversos outros problemas foram encontrados. Muitas vezes, encontramos uma ideia interessante, mas não encontramos uma maneira de implementá-la, pois muitas informações e recursos de que necessitávamos estavam ora indisponíveis, ora extremamente desatualizadas (como por exemplo, recursos para o jogo DooM original, onde encontramos sites de antes do ano 2000). Nos jogos onde conseguimos encontrar as informações necessárias, descobrimos que muitos deles não eram compatíveis com o método de interface do OpenTuner, que interage com outros programas via entrada e saída de linha de comando.

Ainda no quesito de se escolher o jogo e o que seria otimizado, passamos também por problemas de ordem conceitual. A primeira ideia que tentamos seguir era a de otimizar um gerador de mapas. O problema aqui é como derivaríamos uma avaliação do mapa gerado de uma maneira que o OpenTuner pudesse avaliar se determinado mapa era melhor ou pior que outro mapa gerado. Nesta época, tivemos a ideia de realizar uma "otimização dupla", em que ajustaríamos o gerador de mapas, e o avaliaríamos com uma IA jogando uma partida de teste no mesmo. Por complexidade demasiada deste método, a ideia foi abandonada.

Por fim, outro desafio foi a parte do trabalho que envolveu pesquisa teórica. Sendo uma área de pesquisa tão nova e inexplorada, não existem muitos outros artigos ou fontes no assunto. Na verdade, não conseguimos encontrar outro trabalho que relacione *autotuning* usado em jogos digitais da maneira que fizemos. Entre os trabalhos relacionados que foram encontrados, tivemos alguma dificuldade inicial em compreendê-los completamente, pois é um assunto razoavelmente complexo.

6.2.2 Disciplinas Relacionadas

Princípios de Desenvolvimento de Algoritmos, como a matéria que introduziu pensamento lógico e me ensinou a pensar em algoritmos.
Laboratórios de Programação 1 e 2, como as matérias que cobriram diversos aspectos uti-

lizados na confecção deste trabalho, como git, LaTeX, programação orientada a objetos, e programação de jogos digitais.
Leitura Dramática, por me ensinar os fundamentos de falar em público, sempre necessário para realizar uma boa apresentação. Programação Concorrente, por me desafiar a aprender a pensar em paradigmas de concorrência e programas mais complexos; o que ajudou com a compreensão de como os sistemas utilizados aqui funcionam.

Capítulo 7

Trabalhos Futuros

Futuramente, uma possibilidade seria verificar a aplicabilidade das técnicas de ajuste fino durante o desenvolvimento de um jogo, projetado desde o início com isso em mente, e avaliar as consequências positivas e negativas decorrentes. Também poderia haver uma exploração de potenciais alvos de otimização, e seu possível impacto sobre desenvolvimento e design do jogo.

Referências Bibliográficas

Ansel *et al.***(2009)** Jason Ansel, Cy Chan, Yee Lok Wong, Marek Olszewski, Qin Zhao, Alan Edelman e Saman Amarasinghe. *PetaBricks: a language and compiler for algorithmic choice*, volume 44. ACM. Citado na pág. 4

Ansel *et al.***(2014)** Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly e Saman Amarasinghe. Opentuner: An extensible framework for program autotuning. Em *Proceedings of the 23rd international conference on Parallel architectures and compilation*, páginas 303–316. ACM. Citado na pág. 1, 4, 5, 6

Bruel *et al.*() Pedro Bruel, Marcos Amarıs e Alfredo Goldman. Autotuning gpu compiler parameters using opentuner. Citado na pág. 1

Fialho *et al.***(2010)** Álvaro Fialho, Luis Da Costa, Marc Schoenauer e Michele Sebag. Analyzing bandit-based adaptive operator selection mechanisms. *Annals of Mathematics and Artificial Intelligence*, 60(1-2):25–64. Citado na pág. 5

Hoos(2012) Holger H Hoos. Programming by optimization. *Communications of the ACM*, 55(2):70–80. Citado na pág. 1, 3

Tiwari *et al.***(2011)** Ananta Tiwari, Jeffrey K Hollingsworth, Chun Chen, Mary Hall, Chunhua Liao, Daniel J Quinlan e Jacqueline Chame. Auto-tuning full applications: A case study. *International Journal of High Performance Computing Applications*, página 1094342011414744. Citado na pág. 3